Digital Investigation 11 (2014) 3–19

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Identifying back doors, attack points, and surveillance mechanisms in iOS devices

Jonathan Zdziarski

32 West Dr., Bedford, NH 03110, USA

ARTICLE INFO

Article history:
Received 10 December 2013
Received in revised form 23 January 2014
Accepted 26 January 2014

Keywords:
Forensics
Exploitation
iOS
Back doors
Security
Malware
Spyware
Surveillance

ABSTRACT

The iOS operating system has long been a subject of interest among the forensics and law enforcement communities. With a large base of interest among consumers, it has become the target of many hackers and criminals alike, with many celebrity thefts (For example, the recent article "How did Scarlett Johansson's phone get hacked?") of data raising awareness to personal privacy. Recent revelations (Privacy scandal: NSA can spy on smart phone data, 2013; How the NSA spies on smartphones including the BlackBerry) exposed the use (or abuse) of operating system features in the surveillance of targeted individuals by the National Security Agency (NSA), of whom some subjects appear to be American citizens. This paper identifies the most probable techniques that were used, based on the descriptions provided by the media, and today's possible techniques that could be exploited in the future, based on what may be back doors, bypass switches, general weaknesses, or surveillance mechanisms intended for enterprise use in current release versions of iOS. More importantly, I will identify several services and mechanisms that can be abused by a government agency or malicious party to extract intelligence on a subject, including services that may in fact be back doors introduced by the manufacturer. A number of techniques will also be examined in order to harden the operating system against attempted espionage, including counter-forensics techniques.

Introduction

German news outlet Der Spiegel ran an article (Privacy scandal: NSA can spy on smart phone data, 2013) in September 2013 citing leaked NSA documents that boasted of the agency's capabilities in hacking iPhones as early on as 2009. As the article describes it, the NSA allegedly hacks into the desktop machine of their subjects and then runs additional "scripts" that allow them to access a number of additional "features" running on the subjects' iPhones. From the article:

> "*The documents state that it is possible for the NSA to tap most sensitive data held on these smart phones, including contact lists, SMS traffic, notes and location information about where a user has been. …In the internal documents, experts boast about successful access to iPhone data in instances where the NSA is able to infiltrate the computer a person uses to sync their iPhone. Mini-programs, so-called "scripts," then enable additional access to at least 38 iPhone features.*"

Another article (How the NSA spies on smartphones including the BlackBerry) from Der Spiegel goes into greater detail, providing examples of instances where a user's photo album and backup data were accessed. Of course, some of this data could have easily been extracted from other possible NSA activities, such as iCloud interception (How the NSA cracked the web), SMS interception (iPhone users are all zombies), or copied from a desktop backup on a compromised computer (Zdziarski; How the

E-mail address: jonathan@zdziarski.com.

http://dx.doi.org/10.1016/j.diin.2014.01.001
1742-2876/

NSA spies on smartphones including the BlackBerry), but given the nature of the article, I'll assume that in at least some circumstances, the NSA appears to claim the capability to access data on the device directly. The scope of this document will be limited only to harvesting data at rest on the device.

While the actual methods could only be confirmed by the agency itself, the simplest, and most technically feasible explanation, based on the techniques described and the data purportedly stolen, is that the NSA has exploited the trusted relationship between a user's desktop machine and a connected iOS device, used that trusted relationship to then start up a number of otherwise protected, yet undocumented data services running on the device, which will be explored later in this paper. This technique provides not only the ability to transfer a significant amount of data from an iPhone (possibly copying to some remote command-and-control server), but would also allow an agency such as NSA to bypass certain aspects of file system encryption, backup encryption, and also (if they chose) perform a number of other capabilities, including the following.

- Download large amounts of decrypted personal information
- Install spyware on the mobile device itself
- Sniff the network traffic going through the device
- Install mobile APN profiles to redirect Internet traffic to a proxy server
- Generate additional pairing records for exclusive use
- Access the content of any application's sandbox
- Perform these and other tasks *without* the user's knowledge

While Der Spiegel made no written mention of WiFi, the technical possibility exists that NSA (or any other malicious attacker) could also do much of this wirelessly, while the phone is sitting in the targeted subject's pocket, or even while they use the device, without any visual indicators. This paper will identify some of the services and poorly protected mechanisms that can be abused to accomplish all of this, and provide some solutions to disable them, with very little notice to the average end-user.

My own experience in researching iOS has led me to believe that Der Spiegel's article is not far off from the same approach I am describing, as recent versions of iOS (including iOS 6 and 7) have seen a lot of new activity in the development of undocumented services to copy very specific personal data items, explained further in this document.

While the consumer has seen new security mechanisms introduced over time, such as backup encryption, new workarounds have also seemingly been added by Apple to work around them (such as adding more undocumented data sources that bypass encryption, explained in Section 3.7). Similar security enhancements have been made to improve the security of iOS 7, such as a confirmation dialog to trust new desktop connections. Unfortunately, this doesn't help when dealing with a compromised desktop that has already established a trusted relationship. To further threaten the effectiveness of this new feature, Apple's new over-the-air (OTA) supervision and automatic enrollment for iOS 7's MDM (iOS 7 and business) makes it much easier for an agency that specializes in hacking to turn a one-time opportunity to connect to a device into long-term surveillance, using new undocumented security bypasses. One such bypass added to iOS 7 (presumably added for enrolled enterprise devices) provides an override for passcode and fingerprint authentication. Such enterprise-grade data assurance features are an easy target for skilled individuals with trusted access to a device. While the hacks of the past have found ways to brute force PIN codes and unravel encryption, new features like this appear to be added to intentionally bypass a number of security features under the right circumstances. The problem, however, is that such privileged mechanics can be taken advantage of in the wrong circumstances when dealing with an adversary within government.

Pairing: the keys to everything

In order to understand how an attacker could penetrate an iPhone from the owner's desktop computer, it's important to understand how pairing works (A cross-platform software library and tools to communicate with iOS devices natively); A pairing is a trusted relationship with another device, where the client device is granted privileged, trusted access. In order to have the level of control to download personal data, install applications, or perform other such tasks on an iOS device, the machine it's connected to must be paired with the device. This is done through a very simple protocol, where the desktop and the phone create and exchange a set of keys and certificates. These keys are later used to authenticate and establish an encrypted SSL channel to communicate with the device. Without the correct keys, the attempted SSL handshake fails, preventing the client from obtaining privileged access.

A copy of the keys and certificates are stored in a single file, both on the desktop machine and on the paired mobile device. The pairing file is never deleted from the device except when the user performs a restore or uses Apple's "Erase All Content and Settings" feature. In other words, every desktop that a phone has been plugged into (especially prior to iOS 7) is given a skeleton key to the phone. This pairing record allows either the desktop, or any client who has copied the file, to connect to the subject's mobile device and perform a number of privileged tasks that can access personal data, install software, analyze network content, and so on. This one pairing file identifies someone as the owner of the phone, and with this file gives anyone trust and access as the device's owner. There are a few frightening things to know about the pairing mechanism in iOS.

- Pairing happens automatically, without any user interaction (up until iOS 7), and only takes a few seconds. Pairing can be performed by anything on the other end of the USB cable. The mobile device must either have no passcode, or be unlocked. If the user has "Require

Passcode" set to anything other than "Immediate", then it is also possible to pair with the device after it is turned off, until the lock timer expires. So if the user has a device unlocked to play music, and connect it to an alarm clock or a charger running malicious code, whatever it's connected to can establish a pairing record that can later on be used to gain access to the device, at any point in time, until the device is restored or wiped.

- While the pairing process itself must take place over USB (Renard), at any time after that, the phone can be accessed over either USB or WiFi regardless of whether or not WiFi sync is turned on. This means that an attacker only needs a couple of seconds to pair with a device, and can then later on access the device to download personal data, or wreak other havoc, if they can reach it across a network. Additionally, an attacker can easily find the target device on a WiFi network by scanning TCP:62078 and attempting to authenticate with this pairing record. As the pair validation process is very quick, sweeping a LAN's address space for the correct iOS device generally only takes a short amount of time.
- Because of the way WiFi works on these devices, an attacker can take advantage of the device's "known wireless networks" to *force a phone* to join their network when within range, so that they can attack the phone wirelessly. This is due to iOS' behavior of automatically joining networks whose name (not MAC address) it recognizes, such as "linksys" or "attwifi". It may even be possible for a government agency with privileged access to a cellular carrier's network to connect to the device over cellular (although I cannot verify this, due to the carrier's firewalls).

Essentially, that tiny little pairing record file is the key to downloading, installing, and even manipulating data and applications on the target device. That is why I have advised law enforcement agencies to begin seizing desktop machines, so that they can grab a copy of this pairing record in order to unlock the phone; a number of forensic imaging products (including some I've written), and even open source tools (A cross-platform software library and tools to communicate with iOS devices natively) are capable of acquiring data from a locked mobile device, so long as the desktop's pairing record has been recovered. The pairing record also contains an escrow keybag, so that it can unlock data that is protected by data-protection encryption (Renard). This is good news for the "good" cops, who do crazy things like get warrants; it's very bad for anyone who is targeted by spy agencies or malicious hackers looking to snoop on their data.

High value services running under iOS

When a user's desktop computer establishes a connection to an iOS device, it talks to a service named *lockdownd*. This runs on port 62078 (Renard; Usbmuxd), and can accept connections across either USB (via Apple's usbmux (Usbmux) protocol), or WiFi via TCP. The lockdownd process acts much like an authenticated version of *inetd*, where the client requests services, which are farmed out to a number of daemons started on the device.

When a client has connected to the iOS device on this port, lockdown forces the client to authenticate by using a host identifier and keys from the pairing record file we've just discussed, issuing a *StartSession* request. On a Mac running OS X, pairing records are often stored in */var/db/lockdown* or *~/Library/Lockdown*. On Windows 7, They're in *C:|Program Data|Apple|iTunes|Lockdown*. Other operating system variants will vary. Once the desktop authenticates, the keys in the file are used to establish an SSL session with the device (A cross-platform software library and tools to communicate with iOS devices natively; 25C3: hacking the iPhone, 2008); the desktop machine is then able to requests any number of services to be started on the phone, using a *StartService* request, identifying the name of the service to be started (A cross-platform software library and tools to communicate with iOS devices natively; 25C3: hacking the iPhone, 2008).

Available services, or as the Der Spiegel article (Privacy scandal: NSA can spy on smart phone data, 2013) calls, "features", include everything from basic backup and sync services, to more suspicious services that shouldn't ever be running on a mobile device, but come preinstalled by Apple's factory firmware.

A file on the device named *Services.plist* provides a catalog of services that can be started by lockdownd (Miller et al.); users who have installed a jailbreak onto their phones can access this file in */System/Library/Lockdown/Services.plist* or it can be copied by decrypting the file system disk of an Apple firmware bundle (xpwntool). Additionally, when a device is enabled for developer mode, a number of other services are added to the */Developer* folder on the device, to allow Xcode to perform tasks such as debugging. The standard catalog of services includes (among others) the following.

It is important to note that the services to follow are available on every iOS device, regardless of whether or not the phone is enabled for development.

com.apple.mobilebackup2 (Renard; A cross-platform software library and tools to communicate with iOS devices natively; Bédrune and Sigwald)

Used by iTunes to create a backup of the user data on the device. This is the most popular service to be cloned by forensics software, as it obtains a majority of user databases, such as address book, SMS, call history, and so on. This is the only service that is affected by turning on iTunes' *Backup Encryption* feature. When backup encryption is on, these files are encrypted, requiring knowledge of the user password in order to decipher. Devices without backup encryption stand to leak a significant amount of information from this service. If a user's desktop is compromised, it is conceivable that a key logger could potentially log the backup password, also putting their encrypted backups at risk. The encryption scheme is publicly documented, and source code to decrypt a backup can be found at Bédrune and Sigwald.

Even though a user may not maintain a local backup on their desktop machine, this backup service can be

connected to with a trusted connection to generate a backup on-the-fly. Earlier versions of this service caused the device to present a modal status screen to the user, however newer versions of iOS merely only display a small sync indicator in the task bar.

com.apple.mobilesync (Renard; A cross-platform software library and tools to communicate with iOS devices natively)

Also used by iTunes to transfer address book, Safari bookmarks, notes, and other information that the user has selected to sync with their desktop machine. This service is not affected by backup encryption, and so clear text copies of personal data will come across this service. Only data that is specifically designated to sync will be transferred by this service.

Earlier versions of this service caused the device to present a modal status screen to the user, however newer versions of iOS merely only display a small sync indicator in the task bar. This service, and the backup service just described, are the only two services that present a visual indicator of any kind to the user when the service is being accessed.

com.apple.afc (Renard; A cross-platform software library and tools to communicate with iOS devices natively)

This service is often used to access the user's camera reel, photo album, music, and other content stored in the/*var/mobile/Media* folder on the device. By communicating with this service, any trusted device can download the entire media folder. Users who have installed a jailbreak on their iOS devices may also notice a *com.apple.afc2* service installed by the jailbreak tool (The iPhone wiki), which allows a trusted desktop machine to access and download *the entire file system*. This service presents no visual indication to the user that the device is being accessed.

com.apple.mobile.installation_proxy (Renard; Evasi0n jailbreak; A cross-platform software library and tools to communicate with iOS devices natively)

This service is invoked whenever iTunes installs an application on a mobile device. With knowledge of how to speak this protocol, malicious software can also use this service to install software on the device. While all software has to be signed by Apple in order to successfully install, any entity with an enterprise developer license can sign code and install it through this service ad-hoc, without distributing it through the App Store, and without registering the device with the Apple Developer Connection. This service presents no visual indication to the user that the device is being accessed.

com.apple.mobile.house_arrest (Renard; A cross-platform software library and tools to communicate with iOS devices natively)

This service can be used to access the contents of any App Store application's sandbox (where user databases, screenshots, and other content for each application is stored). By iterating through the list of installed applications on the device, it is possible to download user databases, suspend screenshots, and other personal information from every third party app on the device. It is also possible to upload content into the application's sandbox, allowing someone to inject data. This service presents no visual indication to the user that the device is being accessed.

While this service is used by iTunes to copy documents in and out of applications, the service itself allows access to persistent application data (that is, databases, caches, screenshots, and the like), allowing forensic tools to recover persistent data that is not normally included in a device backup.

com.apple.pcapd (Hacking iOS applications)

Connecting to this service immediately starts a packet sniffer on the device, allowing the client to dump the network traffic and HTTP header data traveling into and out of the device. While a packet sniffer can, on rare occasion, be helpful to a developer writing network-based applications, this packet sniffer is installed by default on all devices and *not* only for devices that have been enabled for development. This means anyone with a pairing record can connect to a target device via USB or WiFi and listen in on the target's network traffic. It remains a mystery why Apple decided that every single recent device needed to come with a packet sniffer. This service presents no visual indication to the user that the device is being accessed.

com.apple.mobile.file_relay (Renard; Evasi0n jailbreak; A cross-platform software library and tools to communicate with iOS devices natively)

The file relay is among the biggest forensic trove of intelligence on a device's owner and, in my best and most honest opinion, a key "backdoor" service that, when used to its full capability, provides a significant amount of that that would only be relevant to law enforcement or spying agencies.

Apple seemingly has been making many changes over the past few years to enable the extraction of information through the undocumented file relay service that really only has relevance to purposes of spying and/or law enforcement. This can be seen by comparing the object code from different operating system versions over time, using a disassembler or other similar tools. For example, early versions of iOS provided a very limited number of data sources, serving primarily diagnostic mechanisms to transfer log files and limited personal data. Newer versions of iOS, however, include a number of additional data sources that more deeply expose private information, including metadata that would be considered useless, even for diagnostic purposes. The services by which this data is transferred have not shown to be used by any legitimate sync or diagnostic applications manufactured by Apple, and in many cases even bypass Apple's own user backup encryption feature.

To illustrate, the following list of six data sources shows the only sources available from iPhoneOS firmware version 2.0.0 (5A347). While iPhoneOS 2 only provided six undocumented data sources, there are 44 known data sources available as of iOS 7 (explained in more detail in Section 3.7).

Data Sources available as of iPhoneOS version 2.0.0

AppleSupport
Network
WiFi
UserDatabases
CrashReporter
SystemConfiguration

Among some newer sources of data are a mechanism to download an entire metadata disk image of the device (a disk image of the entire file system, including timestamps, file names, file sizes, and more, but without the actual file content), served up through a data source named *HFSMeta*, the retrieval of cached *GPS positioning* data through a special data source to transfer the *locationd* cache, the user's calendar, a dump of the typing cache, device pairing records, all third party application data, and plenty of other data that should simply not be allowed to come off the device without knowledge of the user's backup password. While no one would argue that a user (or an Apple Store) may opt to backup basic information on a device, one must ask why an undocumented service exists to copy the same (and much more) data than the backup conduit already does, but bypasses the manufacturer's own user encryption security mechanism.

The file relay service's job is to accept a list of requested data sources, and deliver a *gzippedcpio* archive of the data requested. Source code to talk to the file relay service has been available since 2009 in the *libimobiledevice* project (A cross-platform software library and tools to communicate with iOS devices natively), however the service has been available since the very first versions of iPhoneOS. No known public software (Xcode, iTunes, Apple Configurator, or others) appear to reference or use the file relay service, and as the data source list grows, it becomes more evident that this service is used to dump large amounts of decrypted personal data (and metadata) from the device, far beyond any diagnostic use (such as an Apple Store), and with capabilities beyond the needs of backup or even software development.

This service completely bypasses Apple's built-in backup encryption, and using the escrow key from a pairing record, information that is normally encrypted is also delivered in clear text. While Apple may have, at some point, tapped a handful of these data sources for diagnostics or other purposes, it appears clear that a majority of this data is far too personal to ever be used by Apple, or for anything other than intelligence and law enforcement applications.

To substantiate this notion, a number of applications from Apple, including iTunes, Xcode, Apple Configurator, Apple Configuration Management Utility, and others, were examined for any traces of the file relay service, and none were found. While one could conceivably attempt to make an argument that such a service could be used for in-store diagnostics, much of the data yielded by file relay is irrelevant to such a use, such as the *HFSMeta* data source, or even services to acquire the user's entire desktop photo album. Additionally, the data is transferred in a raw form that cannot be restored back onto a replacement device, such as how a device backup could, making it useless for in-store device repair or replacement. Lastly, the file relay bypasses user backup encryption – a security mechanism provided by Apple to protect this same data. It would seem grossly inappropriate for an Apple Store to perform work on a device to which the user had not given them a backup password for, as is required for work on desktop machines, or to even be granted private access to a user's personal data without the customer's direct consent, and with their belief that their data was protected by a password.

Data sources that can be requested from file relay follow.

Accounts

A list of accounts configured on the device. This is typically email accounts, although other types of accounts could also exist. Account passwords are not included in this data, but only the account identifiers.

AddressBook

An unencrypted copy of the user's address book and address book photos. These SQLite databases could potentially include deleted records that can be recovered with SQLite forensics tools.

Caches

The user's *Caches* folder, located in*/private/var/mobile/Library/Caches.* This folder contains screenshots taken whenever preloaded applications are suspended, revealing the last thing a user was looking at in each application. They also contain a number of shared images and offline web content caches, contents of the last copy or cut to the clipboard (pasteboard), map tile images, keyboard typing cache, and other records of personal activity.

CoreLocation

GPS positioning logs. This is a cache of the device's GPS locations, taken at frequent intervals both by cellular and WiFi. In iOS 6, the files*lockCache_encryptedA.db* and *cache_encryptedA.db* (neither of which are actually encrypted) appear to be similar to older iOS 4 *consolidated.db* files that caused quite an uproar in the community a few years ago for caching so much data. Some

documents appear to indicate the NSA exploited (NSA spies reportedly exploited iPhone location bug) these. Examining the cache of a personal device running iOS 6, there are over 500 cellular entries with some entries containing locations that haven't been visited in months, and over 3000 WiFi location entries containing SSIDs of neighboring access points including neighbors and nearby businesses. This cache is used to help return a GPS position based on WiFi, however the cache does still appear to hold some historical information, rather than the seven days advertised.

An agency capable of forcing manufacturers to add back doors certainly also has the power to harvest this information every seven days from the device, or possibly from the manufacturer, so the time period for which this data recycles is irrelevant.

EmbeddedSocial

Log files stored in*/var/mobile/Library/Logs/Social*. It is assumed that this contains logs related to iOS' embedded social networking, such as Facebook and Twitter. No further information is available yet, as this is a new service to iOS 7.

GameKitLogs

Game Kit logs including existing games, enrollments, account information, and other GameKit related data and log files.

HFSMeta

A metadata disk image of the entire file system of the device. That is, a disk image with everything except the actual device content. Timestamps, file names, file sizes, creation dates, and other metadata are all stored in this image. This is a new data source added with iOS 7. The image is transmitted as a *sparseimage* format file, which can be mounted on a Mac by simply double clicking it.

Keyboard

Contents of the key cache; also referred to as the *key logger cache*. These dictionaries contain both an ordered list of the most recent things the user has typed into the keyboard in any application, and a complete dictionary and word count of all words that have been typed. Regardless of whether the input was for SMS, Mail, an App Store application, or any other application, the contents of typed correspondence (including that typed into otherwise secure applications) is copied into this cache in the order in which it was typed (Zdziarski and Media).

Lockdown

A copy of all pairing records (and their escrow bag) stored on the device. This can be used to determine how many and which computers a device has been synced or paired with. The dump also contains a copy of the data ark (a large registry of device information and general operating parameters) and activation record. This data can be used to determine when the device was last activated (evidence of a wipe, or restore), and determine other device settings such as development state, backup encryption, original setup time, device name, time zone, the hostname and OS of the last desktop to backup the device, and other settings. Additionally, if a pairing is permitted while the device is locked, escrow keybags can be recovered from this service to potentially unlock data-protect enabled items from other data sources and services.

MobileCal

A complete database dump of the user's calendar and alarms, as well as log files and preferences. These files are in SQLite database format, allowing deleted entries to possibly be recovered with SQLite forensics software.

MobileNotes

A copy of the user's notes database, where everything is stored in the Notes application, and the user's note preferences. These files are in SQLite database format, allowing deleted entries to possibly be recovered with SQLite forensics software.

Photos

A copy of the user's entire photo album stored on the device.

SystemConfiguration

Contains a WiFi access point cache, containing timestamps and SSIDs of known networks and the last time they were joined. Also contains information about configured accounts network interfaces, and other configuration information. This can, among other things, be used to place the device on a given WiFi network at a given time. If future iOS devices support fingerprint scanning, it could place an individual at the scene of a crime, and not just their mobile device.

Ubiquity

Contains diagnostics information about iCloud, including information about peers that data is shared with. There is also a chunk store database, which may contain sensitive cached data. Conflicts created during iCloud syncs appear to be logged as well, potentially leaking metadata about the user's iCloud content.

UserDatabases

A copy of the following user databases on the device: address book, calendar, call history, SMS database, and email metadata (envelope index). These files are in SQLite database format, allowing deleted entries to possibly be recovered with SQLite forensics software.

VARFS

A virtual file system metadata dump in statvfs format; this will likely be phased out in the future, given the new *HFSMeta* data source. The structure for this appears to be as follow.

```
4 byte header structure (appears to be zeroes)

0x2c bytes
struct statvfs {
unsigned long   f_bsize;  /* File system block size */
unsigned long   f_frsize; /* Fundamental file system block size */
fsblkcnt_t      f_blocks; /* Blocks on FS in units of f_frsize */
fsblkcnt_t      f_bfree;  /* Free blocks */
fsblkcnt_t      f_bavail; /* Blocks available to non-root */
fsfilcnt_t      f_files;  /* Total inodes */
fsfilcnt_t      f_ffree;  /* Free inodes */
fsfilcnt_t      f_favail; /* Free inodes for non-root */
unsigned long   f_fsid;   /* Filesystem ID */
unsigned long   f_flag;   /* Bit mask of values */
unsigned long   f_namemax;/* Max file name length */
}

foreach file {
       4 bytes size_t of filename to follow
       N bytes filename

       4 bytes size_t fts_parent (fts_pointer)
       0x10 bytes struct FTSENT->fts_name[0] + 60

       Possibly (struct FTSENT + 60):

       void *fts_pointer;              /* local address value */
       struct ftsent *fts_parent;      /* parent directory */
       struct ftsent *fts_link;        /* next file structure */
       struct ftsent *fts_cycle;       /* cycle structure */
       struct stat *fts_statp;         /* stat(2) information */
}
```

Voicemail

The user's voicemail database and audio files stored on the device. Voicemail files are stored as AMR audio files; a SQLite database provides information about call duration, and caller metadata.

Other Data Sources that can be requested from the File Relay:

AppleSupport	Apple support logs
AppSupport	The *com.apple.AppSupport.plist* configuration, containing country codes for home and network countries
AppleTV	Apple TV playback logs. This suggests the data is not intended for use in an enterprise environment.
Baseband	Baseband diagnostics logs
Bluetooth	Bluetooth server logs
CrashReporter	Application crash logs, typically submitted to Apple
CLTM	The files*/var/logs/cltm.log* and*/var/logs/tGraph.csv*. Not much is known about these files, as they do not exist on devices tested.
DataAccess	Data access and migration diagnostics logs; appears to be a list of accounts data is imported from, and possibly metadata.
DataMigrator	Data migrator diagnostics logs
demod	Unknown
Device-o-Matic	Device-o-Matic logs; Unknown
FindMyiPhone	Find-my-iPhone logs. This data source is new to iOS 7.
itunesstored	Logs documenting the environment of the user's iTunes store experience.
IORegUSBDevice	IORegistry information. This is empty in iOS 7.
MapsLogs	Map logs and query information, including the *NavTraces* folder.
MobileAsset	Installed asset configurations, such as text input certificates and information, installed dictionaries, and other related information.
MobileBackup	Mobile backup agent logs containing miscellaneous diagnostic information from the backup agent.
MobileDelete	This is a new data source for iOS 7 and appears to store block cleanup logs, created by a housekeeping service named *librarian*.
MobileInstallation	A complete list of installed applications, as well as a log containing information about applications that have been previous installed/uninstalled.
MobileMusicPlayer	Contains airplay logs, including a list of airplay devices that are available on the network.
NANDDebugInfo	Very basic FTL (flash translation layer) debug info.
Network	PPP networking logs
SafeHarbor	Returns the contents of the mobile user's *SafeHarbor* folder, which appears to be empty on all test devices. This may be related to Cisco's VPN networking.
tmp	Contents of the/tmp folder on the file system, which is outside the sandbox and can sometimes include sensitive information.
VPN	Seemingly phased out, and combined with other data sources, used to contain VPN logs.
WiFi	General WiFi logs
WirelessAutomation	Log files for coreautomationd

Other forensically relevant services

Other services yielding good forensic data on the device include the following. All of these could potentially be abused by an individual, agency, or malicious software with a valid pairing record.

com.apple.iosdiagnostics.relay

Provides detailed network usage per-application, on a per-day basis. The data provided details the amount of network usage in Kilobytes for the past several days, grouped by application identifier. This can be used to show the activity of a particular user over a period of time, which applications they transfer the most data from, and can help to correlate potential evidence involving data transmitted over a network.

com.apple.mobile.MCInstall

This service can be used to install managed configurations, including those that contain additional data assurance privileges, such as the ability to bypass certain security mechanisms, locate the device using GPS, or wipe the device.

com.apple.mobile.diagnostics_relay

Includes additional diagnostics information about the device, such as battery and hardware state.

com.apple.mobile.heartbeat

Used to maintain a wireless connection to lockdown and any services accessed. This service essentially performs a simple heartbeat, and will cause all connectivity to the destination to freeze unless a regular heartbeat is received.

com.apple.syslog_relay

Can be used to download and stream a device's system log, and many others.

Installation of invisible, malicious software

Malicious software does not require a device be jailbroken in order to run. An attacker with the right know-how and 20–30 s with a trusted mobile device can install malicious software capable of spying on the user. As was demonstrated at this year's Black Hat 2013 conference in Las Vegas, the Mactans presentation (Lau et al.) revealed what many in the community have known for quite some time, but avoided discussing. With the simple addition of an *SBAppTags* property to an application's *Info.plist* (a required file containing descriptive tags identifying properties of the application), a developer can build an application to be hidden from the user's GUI (SpringBoard). This can be done to a non-jailbroken device if the attacker has purchased a valid signing certificate from Apple. While advanced tools, such as Xcode, can detect the presence of such software, the application is invisible to the end-user's GUI, as well as in iTunes. In addition to this, the capability exists of running an application in the background by masquerading as a VoIP client (How to maintain VOIP socket connection in background) or audio player (such as Pandora) by adding a specific *UIBackgroundModes* tag to the same property list file. These two features combined make for the perfect skeleton for virtually undetectable spyware that runs in the background. If the device is rebooted, applications

tagged as VoIP applications are automatically invoked to handle reconnection.

When an application runs in the background, it can request up to 10 min of background execution time. There is an exception to this, however, and that is for applications that exhibit a form of activity through a VoIP socket or while playing audio. An application need only establish a socket with a remote server, and label the socket a VoIP socket, in order for the application to receive execution cycles automatically whenever data is pushed through the socket. Even if the application is suspended in the background, a remote server transferring as little as one byte of data through an open VoIP socket will cause the application to become active again. This technique makes it possible to receive virtually unlimited background cycles, albeit in chunks of 10 min. Marking a socket as a VoIP socket can be done using Apple's built-in read and write stream property statements.

```
CFReadStreamSetProperty(readStream, kCFStreamNetworkServiceType,
kCFStreamNetworkServiceTypeVoIP);
CFWriteStreamSetProperty(writeStream, kCFStreamNetworkServiceType,
kCFStreamNetworkServiceTypeVoIP);
```

Based on the Der Spiegel article, an agency such as the NSA could easily use the device's mobile installation service to sign a malicious application with an enterprise certificate and install it on any target's device *without* ever registering that device through Apple's developer program (How: Gameboy Emulator finding it's way onto non-jailbroken devices; A cross-platform software library and tools to communicate with iOS devices natively).

Such an application could include a payload of transmitting personal data, taking screenshots, recording audio, obtain geo-location information, or performing a number of other spying tasks (Seriot, 2010). Additionally, the software could be programmed to attack the device to which it is running on, to obtain escalated privileges or other sensitive information. In fact, Apple has recently beefed up this type of security as of iOS 7 to prevent a device from connecting to its own lockdown port, as was allowed in all prior versions of iOS. Lastly, an application running on a user's device could be manipulated to attack other devices on the network, either with exploit frameworks, or by attempting to connect to other iOS devices' lockdown ports over WiFi, if the agency had previously copied pairing records from a subject's desktop machine. While there is no evidence to show that this has ever occurred, the possibility exists for an agency that can infiltrate desktop machines to install software that could attack the device it is running on, as well as all other devices on networks that it joins. This makes a tempting attack point for an agency looking to infiltrate a high value target's known WiFi networks, or to attack associated targets.

A very simple skeleton includes only two changes to the application's *Info.plist* file in order to make this possible (Lau et al.; How to maintain VOIP socket connection in background).

```
<key>SBAppTags</key>
<array>
<string>hidden</string>
</array>

<key>UIBackgroundModes</key>
<array>
<string>voip</string>
</array>
```

At the very bare minimum, a payload could be programmed to execute every 10 min with the simple addition of a VoIP connection handler. The handler would fire every 10 min, even if the device were rebooted (How to maintain VOIP socket connection in background).

```
[[UIApplication sharedApplication] setKeepAliveTimeout: 600 handler:^(void)
{
        /* insert payload here */
}
```

An application using this skeleton would be virtually undetectable by the average user. It would not appear in iTunes or on the user's SpringBoard. Additionally, a managed configuration could be loaded onto the device to prevent the deletion of applications should it ever be discovered. The only way to detect this type of application is to use Xcode's developer tools to browse the applications installed in the device's sandbox, of which such an application would be listed.

If the device is jailbroken, malware can be made much more invisible and elusive, embedded into operating system components, and virtually undetectable even to some experts.

Fingerprint/passcode pairing security override

Introduced with iOS 7 was a new security mechanism to help thwart juice jacking. This was a good step forward in terms of pairing security, and ensured that any device attempting to establish a trusted relationship with the device had to be explicitly authorized by the user, through

means of tapping a "Trust" button. With this feature, plugging an iOS device into someone's laptop, or a malicious charger or other device, would cause a confirmation screen to require the user first trust the device before granting privileged access. This feature would, if correctly functioning, prevent an attacker from establishing a new pairing relationship with a targeted device, limiting their capabilities so that they would have to steal an existing pairing record from a desktop machine, which is not always feasible. In addition to this, such a feature would potentially interfere with more clandestine (black bag) approaches to establish pairing with and access high profile targets' devices in the field (such as a hotel or bar), where a paired desktop may not be available, would be limited by iOS 7's new security feature.

With this new feature, and with the introduction of fingerprint locking mechanisms for newer iOS devices (making it difficult to turn over a "password" to an enterprise upon leaving the company), also comes an apparent bypass to the device locking mechanism, which overrides the device's passcode/fingerprint lock and user trust checks completely, allowing the device to be paired, synced, and possibly unlocked later with the user's original escrow bag, which can be obtained through the file relay service on the device. This allows the device to be paired not only without authorization from the user, but also while locked with a passcode or a fingerprint. While this feature remains undocumented as of the date of this paper, it is presumed that the purpose of this bypass mechanism is to service legitimate enterprise owned devices enrolled into a managed profile, so that an employee's device can be forensically acquired after leaving the company, are incapacitated, or unwilling to provide their fingerprint or password to unlock the device. The mechanism which allows the bypass to be engaged performs a check to ensure that the supervisor certificates provisioned on the device match that of the supervisor attempting to bypass the lock. This feature is present in iOS 7's public release.

Apple's new over-the-air (OTA) supervision and automatic enrollment for iOS 7's MDM (iOS 7 and business) would appear to allow enterprise or government-owned devices to be configured out of the box with a set of restrictions upon activation. Additionally, later enrollment into an enterprise MDM could potentially also enable this security bypass mechanism to permit corporate access to any device that has been enrolled.

In short, there appear to be two ways to apply a cloud configuration to an iOS 7 device: through enrolling the device with an enterprise MDM (using an existing paired connection), or over-the-air through Apple's servers, at the time of activation.

Centrally managed cloud configuration

The pairing security bypass is tied to the Managed Configuration (MC) portions of the operating system, which touch, but also operate independently of systems to enroll and manage mobile device management (MDM) restrictions for an enterprise. The actual data to activate the bypass is stored in a class named *MCCloudConfiguration*. A managed configuration can be written through the device's public facing MDM service *com.apple.MCInstall*, with proper pairing authentication, or written directly from code running on the device, such as configuration daemons, or any other process with privileged (root) access.

The managed configuration framework includes a daemon named *teslad*, which has what appear to be direct hooks into Apple's servers for the loading of managed cloud configurations (the configuration containing– among other restrictions – the pairing security bypass). The *teslad* daemon, part of the Managed Configuration framework, downloads a cloud configuration certificate from *https://iprofiles.apple.com/resource/certificate.cer*, and performs a number of different validations against sessions (*https://iprofiles.apple.com/session*) and profiles (from *https://iprofiles.apple.com/profile*) between the device and a configuration service referred to as Absinthe, hosted on Apple servers. The daemon identifies itself with an HTTP User-Agent of *ConfigClient-1.0* to the server.

It is worth noting that a successful MiTM combined with a certificate forgery (both presumed to be within the reach of government agencies such as NSA), that all signing could potentially become compromised. Code already exists in the public to emulate an Apple policy server, which could be used to change the device's policy (The iOS MDM protocol). This may not even be necessary, however, as a suspicious switch appears to already exist to disable certificate verification over the HTTPS connection. The `-[MCTeslaConfigurationFetcher connection:willSendRequestForAuthenticationChallenge:]` method checks for a configuration directive named `MCCloudConfigAcceptAnyHTTPSCertificate`, and if set, will automatically bypass the server trust process, effectively allowing any web server to masquerade as Apple's Absinthe server.

The *teslad* daemon suggests that Apple has the ability to centrally load a managed configuration onto a device, when invoked. The daemon's `-[MCTeslaConfigurationFetcher convertCloudConfigDictionary:toManagedConfiguration:]` method allows data transferred across the network connection as an `NSDictionary` object to be converted into a managed configuration, where it is sent back to the client that requested the configuration. The only client identified thus far is Apple's *Setup* program on the device. Apple's Setup program attempts to set up a cloud configuration when the device is first activated, inside the method named. `-[ActivationController _fetchCloudConfig]`. If a cloud configuration is written when the device is first activated, it can be programmed to later check in at regular intervals for changes to the policy.

Once an MDM is installed, a check-in mechanism is invoked via APNS (Apple Push Notification Service) to apply new management changes (The iOS MDM protocol). The profile can later be updated remotely through a mechanism that pulls a new cloud configuration from a URL. A new cloud configuration can be downloaded by invoking the `-[MCProfileConnection retrieveCloudConfigurationFromURL:username:password:anchorCertificates:completionBlock:]` method. HTTP based managed configuration makes for a delicate attack surface for NSA to target, at best.

Possible uses

While it is likely that Apple has added this feature exclusively for legitimate use by enterprises, even this has some serious implications: with today's BYOD culture, employees may be unknowingly allowing their personally-owned devices to be forensically accessible to a company's internal investigations team (as well as law enforcement, with the enterprise's consent) by simply enrolling it into the corporate MDM environment. Additionally, new employees issued devices may be permitted to retain personal information on their corporate device without first being informed that their devices could, at any time, be subject to a search that bypasses security.

In addition to potential abuse by the enterprise, an agency seeking to commit espionage could potentially set up their own MDM profile and enroll the device from a compromised desktop, using services running under the device's lockdown. The advantage of this would be to take advantage of an otherwise short-term connection with the desktop to enroll the device itself into an MDM. When installed, MDMs can also be configured to be removable only with a passcode.

It is speculative, however worth mentioning, that select law enforcement agencies could potentially also be granted a mechanism to push such a configuration to a device through Apple's configuration server, or through other means, but possibly only at the time the device is provisioned (post purchase or firmware upgrade). Given Apple's recent patent filing to allow certain restrictions to be wirelessly pushed to devices in secure government facilities (Apparatus and methods for enforcement of policies upon a wireless device), it's conceivable that such restrictions overlap with the same managed configuration interfaces. If Apple has developed the capability to push a camera restriction to devices that are not enrolled in an MDM, then it is also possible that they may have developed the capability to push security bypasses as well, for purposes such as InfoSec enforcement at military installations, or under subpoena. No evidence has been found supporting that this has actually occurred, however.

Given recent articles of Apple deluged by requests to image mobile devices for law enforcement (Apple deluged by police demands to decrypt iPhones, 2013), providing limited law enforcement access to such a bypass could be beneficial for Apple, by providing a mechanism to remotely unlock a device for a specific purpose, where it can be forensically acquired by existing commercial tools. The benefit for Apple to do this would be to lighten the load and cost involved with manually processing subpoenas for data acquisition, to which Apple has reportedly been months behind (Apple deluged by police demands to decrypt iPhones, 2013). Again, however, such a mechanism would be required to be pushed from a trusted supervisor, and so its success would greatly depend on how the device was provisioned by *testlad* out of the box, or later provisioned by an enterprise.

Lastly, due to the *MCTestIaConfigurationFetcher* mechanism, which looks for a managed configuration at device setup time, it is interesting to note that, were a government agency (such as NSA) and Apple cooperating on this level, it is plausible that Apple could provision any device as a supervised device fresh out of the box using this mechanism. This would allow for the agency to potentially target a subject in such a way that whenever the subject either upgraded their device or restored an existing one, that it would be automatically provisioned as a supervised device when it was set up. With this kind of collusion, the supervisor could monitor, and even install software on the subject's device without their knowledge or approval, in the same way that the mechanism is seemingly already used to allow this functionality for enterprises looking to perform bulk provisioning.

Anatomy

As mentioned earlier in this article, the *lockdownd* process is responsible for performing all pairing and authentication of new connections, before allowing new services to be spawned. Previous versions of iOS would deny pairing of locked devices with the error *Password-Protected*. Looking at iOS 7's lockdownd daemon, two new branch instructions have been added to bypass this check.

The code segment just shown depicts part of the security check performed inside lockdownd when a pairing is requested. At the beginning of the code segment, a subroutine inside lockdownd (sub_1F100, actually named *mc_allow_pairing*) is called, which in turn invokes the `-[MCProfileConnection hostMayPairWithOptions:challenge:]` method inside of Apple's managed configuration framework. This check results in one of four possible actions.

- Deny all pairing
- Allow pairing, but prompt the user
- Allow pairing with no user prompt (and while locked)
- Allow pairing with a challenge/response

The results are returned into three different variables; one as a return value from the subroutine, and two assigned to variables whose pointers are passed into *mc_allow_pairing* as arguments. These three variables indicate whether pairing is allowed at all, whether pairing security should be completely bypassed, and whether pairing should require a challenge/response.

Between the call to *mc_allow_pairing* at 193A6 and the code at 193D8 (which denies pairing if the device is locked with a passcode, or later on if the user doesn't approve it), are two bypasses. One of these bypasses is significant. The branches to 19B06 (from 193C0) effectively "skips over" pairing security entirely if the call to *mc_allow_pairing* allows pairing with no user prompt. This is determined from instructions within the *hostMayPairWithOptions* method, which obtains information from the device's cloud configuration, and performs a type of X509 certificate based validation to ensure the bypass is authenticated with the proper credentials. The source of this certificate is unknown, but assumed to be related to an

```
      ; Check -[ MCProfileConnection hostMayPairWithOptions:challenge: ]
__text:0001938E                 LDR.W           R0, [R8,#0xC]
__text:00019392                 BL              sub_5754
__text:00019396                 CMP             R0, #0
__text:00019398                 BNE.W           loc_19AA8
__text:0001939C                 LDR.W           R1, [R8,#0x1C]
__text:000193A0                 ADD             R2, SP, #0x7E8+var_420
__text:000193A2                 ADD             R3, SP, #0x7E8+out
__text:000193A4                 MOV             R0, R4
__text:000193A6                 BL              sub_1F100

      ; Pairing is explicitly forbidden by MC
__text:000193AA                 CMP             R0, #0
__text:000193AC                 BEQ.W           loc_19AB0

      ; Pairing is allowed by MC, but with challenge/response
__text:000193B0                 LDRB.W          R0, [SP,#0x7E8+out]
__text:000193B4                 CMP             R0, #0
__text:000193B6                 BNE.W           loc_19AC2


      ; Pairing is allowed by MC while locked / untrusted without
      ; any challenge/response (pairing security is bypassed)
__text:000193BA                 LDRB.W          R0, [SP,#0x7E8+var_420]
__text:000193BE                 CMP             R0, #0
__text:000193C0                 BNE.W           loc_19B06

      ; Pairing is allowed while locked / untrusted if the device
      ; doesn't support it
__text:000193C4                 MOV             R0, #(cfstr_Hasspringboa_1 -
0x193D0) ; "HasSpringBoard"
__text:000193CC                 ADD             R0, PC  ; "HasSpringBoard"
__text:000193CE                 BLX             _MGGetBoolAnswer
__text:000193D2                 CMP             R0, #1
__text:000193D4                 BNE.W           loc_19B06

      ; Actual pairing security routines (check device lock, whether
      ; user has pressed "Trust", and so on)

__text:000193D8                 MOVS            R0, #0
__text:000193DA                 BLX             _MKBGetDeviceLockState
__text:000193DE                 MOV             R3, R0
__text:000193E0                 SUBS            R0, R3, #1
__text:000193E2                 CMP             R0, #2
__text:000193E4                 BCS.W           loc_19ADE
__text:000193E8                 LDR.W           R0, [R8]
__text:000193EC                 MOVS            R1, #1
__text:000193EE                 BL              sub_14FE0
__text:000193F2                 MOVS            R6, #0
__text:000193F4                 B.W             loc_19EF2
...
```

MDM certificate. Once the certificate passes validation, the bypass is enabled. When this occurs, the lock state of the device is never checked, and the user is never prompted to trust the host it is connected to. The code branches to the same location that the bypass at 193D4 branches to if pairing security is not supported (for example, on an AppleTV or other device without a screen lock).

A look inside of the *ManagedConfiguration* framework inside Apple's shared cache shows that the decision to bypass pairing security is based, in part, on certificate data retrieved from a call to `[ [ MCProfileConnection sharedConnection ] cloudConfigurationDetails ]`.Since iOS 7 does not yet have a jailbreak, it's difficult to determine just what information is stored in this configuration.

If pairing security is overridden through this mechanism, then both the user trust prompt and the device lock test are bypassed, allowing the device to continue pairing, even if a passcode is set. The logic, in pseudocode, works as follows.

```
if (mc_allows_any_pairing == false) { /* MDM prevents any pairing */
    error(PasswordProtected);
}

...

if (mc_allows_pairing_while_locked || device_has_no_springboard_gui) {
    goto skip_device_lock_and_trust_checks; /* Skip security */
}

/* Pairing Security */

if (device_is_locked == true) {
    if (setup_has_completed) {
        if (user_never_pushed_trust) {
            error(PasswordProtected);
        }
    }
}

/* Bypass ... */

skip_device_lock_and_trust_checks:
    ... pairing process continues (validate host challenge, etc)
```

At it's very best, the device security bypass is an undocumented MDM feature allowing enterprises to access any enrolled (or over-the-air enrolled) iOS MDM device. Even this, however, creates a significant threat to the security of the many iOS users working for companies with a BYOD policy.

Suspicious design omissions

While inconclusive, it is worth noting that the iOS operating system has noticeable omissions in the design of the security architecture, which severely degrade the performance of otherwise "good" security implementations. The most notable of these are outlined in this section. Ironically, these design omissions are well within the grasp of the manufacturer to implement, likely with little effort in comparison to the rest of the design. It is unknown whether these omissions are the direct result of pressure from a government agency, or simply poor design choices, however it is difficult to grasp how a team of engineers that are bright enough to have designed such an impressive system could have intentionally missed such significant issues that are detrimental to its security, lending to speculation that their omission may be intentional.

It is often mistaken that the iOS operating system delivers the same level of security as its desktop counterpart (OS X). Such design omissions do not appear in the desktop operating system.

Boot-stage disk key left unprotected

The iOS file system depends on a key hierarchy, whose top tier keys are stored in the effaceable storage (block zero) of the NAND. This allows for quick data destruction; by simply overwriting these top tier keys, the entire file

system becomes unrecoverable. One significant design omission in this mechanism is a means of protecting the top tier keys with a user secret, and those top tier keys protect any files that are not explicitly protected with one of the *NSFileProtectionComplete* classes. Unlike the data-protection class keys, protected with a user passphrase, the top tier keys are protected using a key derived from hardware-based information. This design omission results in a majority of user data from the file system at risk of exposure to an attacker with code-execution privileges or the ability to extract and deduce the hardware-based keys. This ability to gain code execution is what rendered virtually unlimited access to data at rest for all iPhone 4 and older devices. The design places a significant portion of user data at risk from threats including a number of jailbreak and forensics tools, possessors of zero-day code-execution exploits, and the manufacturer (Apple), who has code signing authority to boot such code on the device.

The encrypted iOS file system causes unique file keys to be generated for every file on the device. These unique file keys are then encrypted and protected in one of two ways. Files marked as protected using data-protection are protected with a key from the file system keybag, which must be unlocked either by the user's passphrase, or the escrow key from a backup key (if the device has not been rebooted since it's last unlock). The second method (and how all files not protected with data-protection encryption are protected) is to protect the file with a key stored in the effaceable storage, referred to as the Dkey (or class-4 key).This Dkey is itself encrypted using keys calculated from the device's unique hardware. No publicly available methods have been found to deduce the hardware keys from the device, however a running kernel can decrypt the file system when the device is booted, allowing anyone with code-execution capabilities to access the large portion of the file system that is not protected with data-protection.

It is unknown whether or not the manufacturer has been (or could be) forced, under secret court order, to provide one or more law enforcement branches with the ability to execute a running kernel with custom code, however Apple is known to provide services to law enforcement to provide an image of the file system, sans what Apple deems, "encrypted files", which are believed to refer to files using data-protection.

The ability to decrypt the class-4 portions of the file system provide an attacker the capability of acquiring a significant amount of user data from the device, as only a small portion of the file system typically uses data-protection. To add to this exposure, however, backups of the keybag keys are stored in an escrow record whenever a pairing record is created with a desktop machine, and can be used to unlock the keybag IF the device has not been rebooted since its last device unlock by the user, subsequently decrypting data that would otherwise be protected with data-protection. This escrow bag is accessed in the directory*/private/var/root/Library/Lockdown/escrow_records* on the mobile device, and is only encrypted with the Dkey. With an escrow record, it is possible to unlock the keybag on the device and access the remaining files protected with data-protection, however the lockdownd services running on the device will do this for the user automatically, whenever an escrow bag is supplied in lockdownd requests. Exploiting this requires only that at least one device was ever paired with the mobile device. This technique is not possible unless the device has – at some point since its last reboot – been unlocked, to satisfy the *NSFileProtectionCompleteUntilFirstUserAithentication* protection of the keybag itself. Black bag techniques involving theft, desktop penetration, and other such types of attacks, however may be possible while the device is in this state.

In contrast, Apple's desktop implementation (File Vault) protects the file system key hierarchy with a key that is derived from a user-supplied passphrase on boot. When implemented properly, only a small portion of boot code lies unencrypted on disk, which then accepts a passphrase in order to unlock the rest of the file system. As a result of this design, the disk encryption provided by File Vault is hardware independent (portable), and is instead dependent on a secret supplied by the user.

The more secure full disk encryption implementation provided by File Vault could have easily been incorporated into iOS, in one of two ways. The iOS boot loader (iBoot) could prompt the user for a passphrase whenever the mobile device is rebooted, or an additional key tier could have been added to the operating system firmware in such a way that a key derived from a user passphrase Key_{User} were prompted for at boot, and incorporated into the file system keys of the user data partition of the device. This would have resulted in the root partition (which is, by default, read-only) being readable at the time of boot (via Dkey), and the user data partition being readable only after $f(\text{Dkey}|\text{Key}_{User})$ could be calculated. Since the user data partition is the only partition that either requires or uses disk encryption, the operating system could, today, completely boot and prompt the user for a passphrase prior to mounting any user data.

This design omission has made it possible for Apple to service law enforcement subpoenas to provide a disk image of the user file system, however this has been at the expense of considerable data security, as evidenced by the number of forensics tools capable of performing physical acquisition of past devices, and the expectation that current and future devices will be (or have been) compromised in a similar fashion.

Packet sniffing services not moved to developer mount

When developer-mode services were added to iOS, a number of services were moved to a remote disk that was mounted to install developer tools on the device. By default, these services are not installed or running on the device, but later installed by developers using Xcode, when the "Use for Development" option is selected with a given device. The packet sniffer service (com.apple.pcapd), whose only practical legitimate use is for developer inspection of packet data, was not moved, and is therefore active on every iOS device, including those that have never been configured for developer mode. One can only speculate as to why a service to capture and analyze packet data, either over USB or WiFi, is active on all devices running recent versions of iOS.

Apple can solve this problem immediately by removing the pcap service from the operating system entirely, or installing it only when a device is placed in developer mode.

Lack of adequate pair purging

When iOS 7 shipped, a new trust dialog was added so that a user must authorize all attempts to establish a pairing. This appeared in Apple's change log immediately after Lau et al. was announced, exploiting the lack of pairing security on iOS devices. From 2007 to 2012, however, all iOS devices lacked this basic pairing security function, allowing any malicious device to establish a semi-permanent pairing. Prior demonstrations (Beware of juice jacking) had been given at security conferences, however, showing the concept of "juice jacking" to gain access to a user's data by masquerading a malicious device as a legitimate one (such as an alarm clock). What the public may not have realized at the time was that such techniques allow potentially permanent and unfettered access to user data, over both USB and WiFi. Unfortunately, today users are being trained by third party products (such as FM transmitters, and third party chargers) to press Trust in order to achieve the functionality they desire, even though it should be unnecessary.

In addition to the lack of adequate pairing authentication, even iOS 7's internal lockdown daemon shows evidence of a new mechanism to reset the pairing on the device, however this reset mechanism can only be accessed from a privileged process running on the device itself, and cannot be triggered over USB or WiFi. As of this writing, no operating system component exists to actually take advantage of this mechanism to allow the user to reset the pairing on the device, which could be used to destroy any rogue pairings established on the device. Lack of this mechanism being functional, all device pairings remain on the device until the device is restored.

Apple can solve this problem immediately by adding a feature to iOS' settings to review and selectively (or completely) reset pairing.

Other bugs seem to persist with respect to pairing security. Supervised devices, whose pairing can be disabled, appear to occasionally allow pairing when connected to devices in certain states (such as certain desktop that are still in the process of booting). These bugs have yet to be weeded out by the manufacturer, however could potentially allow pairing even when restricted by corporate policy.

Lack of pairing record pinning

As was noted earlier in Boot-stage disk key left unprotected section, pairing records contain an escrow key capable of unlocking the keybag on the device; these escrow keys are not only stored unencrypted on the device itself, but also stored unencrypted on the desktop machines they are paired with. By failing to wrap the pairing record with any form of user passphrase on either side, attackers compromising desktop machines, such as appears the case with individuals targeted by NSA, are able to copy and steal the pairing relationship with any mobile devices the desktop is paired with.

Apple could solve this problem immediately by providing an optional mechanism to encrypt pairing record keys with a user passphrase, such that a user must enter a passphrase the first time they sync with the device after reboot. If implemented, this would prevent the theft of pairing record data while at rest from both malware and attackers.

Lack of geofencing in fingerprint reader

The iPhone 5s introduced a fingerprint reader as a means of authenticating the user on a device. Because a user's biometrics are not considered protected in the same way as a password by the US Fifth Amendment (as far as the courts are concerned), a user can (in many cases) be legally compelled to authenticate their fingerprint on the device. Additionally, with a few thousand dollars of equipment, the fingerprint reader can be defeated (German hackers allegedly break touch ID fingerprint scanner) with a fingerprint lifted from the device, or another source. Most fingerprint labs already possess the required equipment to manufacture a print. This mechanism, when used, poses a significant risk of those who are targeted by an attacker or a government (including a foreign government) seeking access to the data on the device. While certain safeguards do exist, such as requiring a passcode after 48 h or after reboot, Apple has not included a capability to disable the fingerprint reader when the user leaves a specific radius.

Apple could solve this problem by incorporating a mechanism to add a geofence into the operating system, so that a fingerprint reader will permanently disable until a passcode is supplied, if the user leaves areas they deem as "secure" locations. In order to make this effective, the operation of the fingerprint reader should only be disabled, and never enabled when entering the perimeter of the geofence.

Integrity of iMessage

In October 2013, Quarkslab exposed design flaws (iMessage privacy) in Apple's iMessage protocol demonstrating that Apple does, despite its vehement denial, have the technical capability to intercept private iMessage traffic if they so desired, or were coerced to under a court order. The iMessage protocol is touted to use end-to-end encryption, however Quarkslab revealed in their research that the asymmetric keys generated to perform this encryption are exchanged through key directory servers centrally managed by Apple, which allow for substitute keys to be injected to allow eavesdropping to be performed. Similarly, the group revealed that certificate pinning, a very common and easy-to-implement certificate chain security mechanism, was not implemented in iMessage, potentially allowing malicious parties to perform MiTM attacks against iMessage in the same fashion. While the Quarkslab demonstration required physical access to the device in order to load a managed configuration, a MiTM is also theoretically

possible by any party capable of either forging, or ordering the forgery of a certificate through one of the many certificate authorities built into the iOS TrustStore, either through a compromised certificate authority, or by court order. A number of such abuses have recently plagued the industry, and made national news (Another certificate authority issues dangerous certificates; Digital certificate authority hacked; Mozilla toughens up on CA abuse).

Apple's response to Quartkslab's research has been continued denial of the technical capabilities to or the desire to intercept iMessage traffic, however the technical details of the report have been validated by a number of independent security researchers. Because any organization that is compelled to perform surveillance on its customers is likely also under a court order to keep such activities confidential, the technical revelations alone are enough to potentially damage trust in the confidentiality of Apple's services; with today's secret government surveillance operations, the only way to truly gauge security is by the quality of the technology. In this case, there appear to be not only flaws, but potentially suspicious flaws, further chipping away at Apple's credibility in securing iMessage.

The design flaws in the iMessage protocol are suspicious to the degree that certificate pinning is a feature already built into the iOS operating system for App Store developers, and has been made very easy to implement by Apple, yet is not implemented in Apple's own software. A reasonable amount of Apple documentation even exists to describe the process by which a developer can implement pinning. It was, as it seems, that Apple is not eating their own dog food. The overall design and use of a centrally managed key directory further calls into question the integrity of the iMessage system, as Apple's implementation allows for the most classic form of MiTM to be performed; a technique has been well known in information security for decades.

Quarkslab has introduced a counter-surveillance technique to help mitigate the risk of iMessage surveillance, by monitoring the public keys for changes. The iMITMProtect tool attaches to the *imagent* process and intercepts keys sent by Apple's key server. If a public key ever changes (which should not happen), the tool will alert the user that their communication may be the target of compromise, and will serve up a cached copy of the public key to allow for continued secure communication with the endpoint. This mechanism will identify and help combat most types of MiTM event, except in cases where a key is compromised from the point of initial exchange. Such an attack would only likely be possible if Apple were substituting keys for one or all users from the moment they are first generated. While a good monitoring and counter-surveillance tool, this is not a complete solution.

Apple could greatly improve the overall security of iMessage. A number of instant messaging protocols incorporate perfect forward security (PFS), which can be used to establish encrypted sessions in an untrusted environment, even if one party's keys are exposed. Because Apple hosts the keys for both parties on a centralized server, moving key generation and storage closer to the end-user, as instant messaging application do, can greatly improve the security of iMessage. The Diffie–Hellman key exchange is a well known and accepted protocol for performing cryptographic key exchange over an insecure channel, and is incorporated by PFS. Finally, implementing the certificate pinning mechanism that Apple, themselves, have already provided for developers, would greatly reduce the likelihood of a MiTM attack using a rogue certificate, or other means.

Overall, the design flaws of iMessage appear to be valid, however the question remains of whether these design flaws were actually mistakes in the design, or omissions intentionally left out of the design.

References

A cross-platform software library and tools to communicate with iOS devices natively. http://www.libimobiledevice.org.

Another certificate authority issues dangerous certificates. http://nakedsecurity.sophos.com/2011/11/03/another-certificate-authority-issues-dangerous-certficates/.

Apparatus and methods for enforcement of policies upon a wireless device. http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=/netahtml/PTO/search-adv.htm&r=36&p=1&f=G&l=50&d=PTXT&S1=(20120828.PD.+AND+Apple.ASNM.http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=/netahtml/PTO/search-adv.htm&r=36&p=1&f=G&l=50&d=PTXT&S1=(20120828.PD.+AND+Apple.ASNM.)&OS=ISD/20120828+AND+AN/Apple&RS=(ISD/20120828+AND+AN/Apple).

Apple deluged by police demands to decrypt iPhones. http://news.cnet.com/8301-13578_3-57583843-38/apple-deluged-by-police-demands-to-decrypt-iphones/; May 2013.

Bédrune Jean-Baptiste, Sigwald Jean. iPhone data protection in depth. http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf; http://code.google.com/p/iphone-dataprotection/.

Beware of juice jacking. http://krebsonsecurity.com/2011/08/beware-of-juice-jacking/.

25C3: hacking the iPhone; 2008; http://theiphonewiki.com/wiki/25C3_presentation_%22Hacking_the_iPhone%22.

Digital certificate authority hacked. http://www.darkreading.com/attacks-breaches/digital-certificate-authority-hacked-doz/231600498.

Evasi0n jailbreak. http://www.evasi0n.com.

German hackers allegedly break touch ID fingerprint scanner. http://www.ibtimes.com/apple-iphone-5s-defeated-german-hackers-allegedly-break-touch-id-fingerprint-scanner-video-1409584.

Hacking iOS applications. http://archive.hack.lu/2012/Mathieu%20RENARD%20-%20Hack.lu%20-%20Hacking%20iOS%20Applications%20v1.0%20Slides.pdf.

How did Scarlett Johansson's phone get hacked?. http://gizmodo.com/5841742/how-did-scarlett-johanssons-phone-get-hacked.

How the NSA cracked the web. http://www.newyorker.com/online/blogs/elements/2013/09/the-nsa-versus-encryption.html.

How the NSA spies on smartphones including the BlackBerry. Der Spiegel. http://www.spiegel.de/international/world/how-the-nsa-spies-on-smartphones-including-the-blackberry-a-921161.html.

How to maintain VOIP socket connection in background. http://stackoverflow.com/questions/5987495/how-to-maintain-voip-socket-connection-in-background.

How: Gameboy Emulator finding it's way onto non-jailbroken devices. http://www.imore.com/how-gameboy-emulator-finding-its-way-non-jailbroken-devices.

iMessage privacy. http://blog.quarkslab.com/imessage-privacy.html.

iOS 7 and business. http://www.apple.com/ios/business/.

iPhone users are all zombies. http://www.theregister.co.uk/2013/09/09/fanbois_the_nsa_thinks_youre_all_zombies/.

Lau Billy, Yeongjim Jang, Chengyu Song, Tielei Wang, Pak ho Chung, Royal Paul. Mactans: injecting malware into iOS devices via malicious chargers. In: Black Hat 2013. Georgia Institute of Technology. https://media.blackhat.com/us-13/US-13-Lau-Mactans-Injecting-Malware-into-iOS-Devices-via-Malicious-Chargers-WP.pdf.

Miller Charlie, Blazakis Dion, Zovi Dino Dai, Esser Stefan, Iozzo Vincenzo, Weinmann Ralf-Phillip. iOS hacker's handbook. Wiley. ISBN 978-1118204122.

Mozilla toughens up on CA abuse. http://news.techworld.com/security/3427196/mozilla-toughens-up-on-ca-certificate-abuse/.
NSA spies reportedly exploited iPhone location bug. http://arstechnica.com/security/2013/09/nsa-spies-reportedly-exploited-iphone-location-bug-not-fixed-until-2011/.
Privacy scandal: NSA can spy on smart phone data. Der Spiegel. http://www.spiegel.de/international/world/privacy-scandal-nsa-can-spy-on-smart-phone-data-a-920971.html; September 7; 2013.
Renard Mathieu. Hacking Apple accessories to pown iDevices. Sogeti. http://www.ossir.org/paris/supports/2013/2013-07-09/ipown-redux.pdf.
Seriot Nicolas. iPhone privacy. http://seriot.ch/blog.php?article=20091203; 2010.
The iOS MDM protocol. http://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf.
The iPhone wiki. http://www.theiphonewiki.com.
Usbmux. http://theiphonewiki.com/wiki/Usbmux.
Usbmuxd. http://theiphonewiki.com/wiki/Usbmux.
xpwntool. http://theiphonewiki.com/wiki/Xpwntool.
Zdziarski. iOS forensic investigative methods. http://www.zdziarski.com/blog/?p=2287.
Zdziarski Jonathan, Media O'Reilly. iPhone forensics. ISBN 978-0596153588.